```
/*
 * IsingLattice.java
 * -----------------------------
 * Separated from Ising.java by David Wong
 */

import java.util.Observable;
import java.util.Observer;

/** Gitter-Klasse fuers Ising-Modell **/
final class IsingLattice extends Observable implements Observer {
  /* Gitterparameter und Feldinhalt */
  private int dim=2;                    // Square and practical
  private int[] ndim;                   // extent in each spatial direction
  private int nind;                     // magnitude of spin field
  private int[][] index;                // field of neighboring indices
  private int[] index0, index1, index2, index3;
                                        // hopefully improved time behavior

  /* Weitere oeffentliche Parameter */
  public short[][] sigma;               // spin field

  private int totalTime = 0;
  public int t=0;                       // actual time

  /* Physikalische Parameter des Modells */
  public static final double TempMin=0.01;
  public static final double TempMax=6;
  public static final double TempCrit=2.2691;
                                        // TempCrit=2/ln(1+sqrt 2)
    private double beta;
  double[] tabexp = new double [9];

  int[] randindex;                      // randomly mutated index field
  IsingIntIndex helpIndex;              // index field for the permutations
  int t0=t, period;                     // time variable
    private Temperature tempValueHolder; // object to keep track of temperature


    void initNDim (int dim, int n) {
        ndim = new int [dim];
        for (int j = 0; j<dim; j++) {
            ndim [j] = n;
        }
    }

    public int getNInd () {
        return nind;
    }

  public IsingLattice(int dim, int n, int period,
                      Temperature tempValueHolder) {
    int i, j;                           // Indexvariable

    /* Gitter initialisieren */
    this.dim=dim;
    initNDim (dim, n);
    this.period=period;
    this.tempValueHolder = tempValueHolder;
    if (!(tempValueHolder.getTemperature () >= TempMin
          && tempValueHolder.getTemperature () < TempMax)) {
        this.tempValueHolder.setTemperature ((TempMin+TempMax)/2);
        // The Golden Mean
    }
```

```
    beta=1/this.tempValueHolder.getTemperature ();
    tabexp[8] = Math.exp(-8*beta) / (Math.exp(-8*beta) + 1. );
    tabexp[6] = Math.exp(-4*beta) / (Math.exp(-4*beta) + 1. );
    tabexp[4] = 0.5;
    tabexp[2] = 1. / (Math.exp(-4*beta) + 1.);
    tabexp[0] = 1. / (Math.exp(-8*beta) + 1.);

    nind=1;                              // Magnitude of the field
    for (j=0; j<dim; j++)
      nind*=ndim[j];

    sigma=new short[period][nind];       // T=2*n: extent in time direction

    index=new int[2*dim][nind];
    randindex=new int[nind];
    for (i=0; i<nind; i++)               // randomly initialize index from 0...nind
      randindex[i]=i;                    // 0..nind initialisieren
    helpIndex=new IsingIntIndex(nind);

    calcIndex();
    index0=index[0]; index1=index[1]; index2=index[2]; index3=index[3];
  }


  /** Coldstart
    value: Initialisierungswert
    **/
  public void initCold(int value) {
    int i;

    for (i=0; i<nind; i++)
      sigma[t][i]=(short)value;
    totalTime = 0;
  }

    public int getTotalTime () {
        return totalTime;
    }

  /** Hotstart **/
  public void initHot() {
    int i, j, m, digit, v, sgn;

    for (i=0; i<nind; i++) {
      v=1; m=i; sgn=0;
      for (j=0; j<dim; j++) {
        digit=m%ndim[j];
        sgn^=digit&1;
        m=(m-digit)/ndim[j];
        v*=ndim[j];
      }
      sigma[t][i]=(short)(2*sgn-1);
    }
    totalTime = 0;
  }

  /** Warmstart **/
  public void initWarm() {
    int i;

    for (i=0; i<nind; i++)
      sigma[t][i]=(short)(2*(short)(2*Math.random())-1);
    totalTime = 0;
  }
```

```
  /** Funktion F mit F(z)/F(1/z)=z **/
  private double F(double z) {
    return (z < 1) ? z : 1;
  }

  /** Indexfelder initialisieren (with periodic BC) **/
  private void calcIndex() {
    int i, j, m, digit, v;

    for (i=0; i<nind; i++) {           // evaluate n. neighboring index
      v=1; m=i;                        // v: "Zwischen"volumen  time magnitude(?)
      for (j=0; j<dim; j++) {
        digit=m%ndim[j];
        index[j][i]=i-(digit-(ndim[j]+digit-1)%ndim[j])*v;
        index[j+dim][i]=i-(digit-(digit+1)%ndim[j])*v;
        m=(m-digit)/ndim[j];
        v*=ndim[j];
      }
    }
  }

  /** fetch value **/
  public short getValue(int x, int y) {
    if (0 <= x && x < ndim[0] && 0 <= y && y < ndim[1])
      return sigma[t][x+ndim[0]*y];
    else
      return 0;
  }

    /* Update the temperature */
    /** Set (or make guess) Temperature **/
    public void update (Observable o, Object arg) {
        double newTemp = tempValueHolder.getTemperature ();
        if (TempMin <= newTemp && newTemp < TempMax) {
            beta=1/newTemp;
        } else if (newTemp > TempMax) {
            tempValueHolder.setTemperature (TempMax);
            return;  // will call update with TempMax indirectly, so
            // return for now
        } else if (newTemp == TempMax) {
            beta=0;
        } else if (newTemp < TempMin) {
          tempValueHolder.setTemperature (TempMin);
          return;
        } else {
            return;
        }
        tabexp[8] = Math.exp(-8*beta) / (Math.exp(-8*beta) + 1. );
        tabexp[6] = Math.exp(-4*beta) / (Math.exp(-4*beta) + 1. );
        tabexp[4] = 0.5;
        tabexp[2] = 1. / (Math.exp(-4*beta) + 1.);
        tabexp[0] = 1. / (Math.exp(-8*beta) + 1.);
    }


    /** 2 dim-Hamiltonian des Ising-Modells
        t: Time **/
  public double Hamilton(int t) {
    int i;
    double sum=0;

    //    System.out.println ("ham");
    for (i=0; i<nind; ++i)
```

```
      sum+=sigma[t][i]*(sigma[t][index0[i]]+sigma[t][index1[i]]);
    //    System.out.println ("ham1");
//   return -2*sum;
     return -sum/nind;
  }

  public double HamiltonMin() {
//  return -2*dim*nind;
    return -dim;
  }

  public double HamiltonMax() {
//  return 2*dim*nind;
    return dim;
  }

  public double Magnetization(int t) {
    int i;
    int sum=0;

    //    System.out.println ("mag");
    for (i=0; i<nind; ++i)
      sum+=sigma[t][i];
    //    System.out.println ("mag1");
    return (double)sum/nind;
  }

  /** Randomly permutate field index **/
  private void perm() {
    int i, k;

    helpIndex.init(randindex[0]);

    for (i=1; i<nind; ++i) {
      k=(int)((i+1)*Math.random());
      helpIndex.insert(randindex[i],k);
    }

    randindex=helpIndex.get();
  }

    private void advanceTime () {
        t=(t + 1)%period;                       // Current time of spin-config
        totalTime ++;
    }

  /** Spin Configuration update for Single-spin-flip dynamics **/
  public synchronized int ssdUpdate() {
    int i, k, de;
    int flag;
    double abuf, rand;
    int sigmasum=0;

    t0=t;                                 // Time of original spin-config
    advanceTime ();

    for (i=0; i<nind; ++i)
      sigmasum+=sigma[t][i]=sigma[t0][i];

    for (i=0; i<nind; ++i) {
      de = 4 + sigma[t][i] *
        (sigma[t][index0[i]] + sigma[t][index1[i]]+
         sigma[t][index2[i]] + sigma[t][index3[i]]);
      //      System.out.println ("jdaf3:"+i);
```

```
      System.out.flush ();
      rand=Math.random();
      //      System.out.println ("jdaf4");
      if ( rand < tabexp[de] ) {
        sigma[t][i]=(short)-sigma[t][i];
        sigmasum+=2*sigma[t][i];
      }
    }


    setChanged ();
    notifyObservers ();
    return sigmasum;
  }
}


/** Index Class is needed for permuting **/
class IsingIntIndex {
  public int[] value;                  // field value
  public int[] prev, next;             // previous/next index
  int n, filled;                       // field magnitude, filling
  int first, last;                     // first/last element
  int i, actual;                       // variable index
  int[] retval;                        // feedback field

  public IsingIntIndex(int n) {        // contructor,  n: field  magnitude
    value=new int[n];
    prev=new int[n];
    next=new int[n];
    this.n=n;
    first=last=-1;
    filled=0;
    retval=new int[n];
  }

  public void init(int val) {          // With first entry initialized
    value[0]=val;
    prev[0]=next[0]=-1;
    first=last=0;
    filled=1;
  }

  public void insert(int val, int k) { // Element value inserted at position k
    actual=first;
    for (i=0; i<k; ++i)
      if (actual > -1) actual=next[actual];
    value[filled]=val;
    if (actual > -1) {
      if (actual!=first) {
        next[prev[actual]]=filled;
        prev[filled]=prev[actual];
      }
      else
        first=filled;
      prev[actual]=filled;
    }
    else {
      prev[filled]=last;
      next[last]=filled;
      last=filled;
    }
    next[filled++]=actual;
  }
```

```
  public int[] get() {                  // Fetch Field
    actual=first;
    for (i=0; i<filled; ++i) {
      retval[i]=value[actual];
      actual=next[actual];
    }
    return retval;
  }
}
```